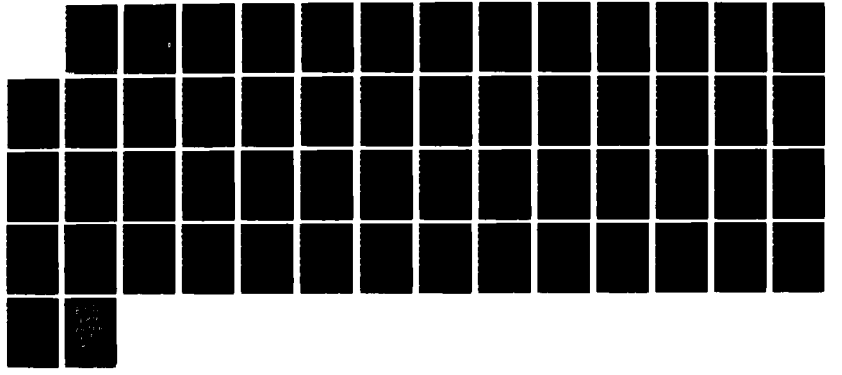
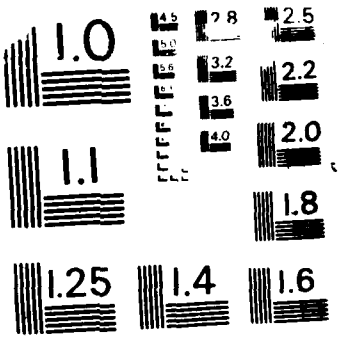
AD-A190 360 ADA (TRADE NAME) COMPILER VALIDATION SUMMARY REPORT: 1/1
DDC INTERNATIONAL DD (U) INFORMATION SYSTEMS AND
TECHNOLOGY CENTER W-P AFB OH ADA VALI 31 OCT 86
UNCLASSIFIED F/G 12/5 NL





DTIC FILE COPY

(2)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. AD-A190 360	2. GOVT ACCESSION NO. / PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: DDC International. DDC Ada Compiler Stystem, Version 4.1, VAX-11/785, VAX-11/750, MicroVAX II, VAX 8200, VAX 8650		5. TYPE OF REPORT & PERIOD COVERED 31 Oct.'86 to 31 Oct.'87 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson AFB, OH 45433066503		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081ASD/SIOL		12. REPORT DATE 31 Oct.'86 13. NUMBER OF PAGES 51
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) Wright-Patterson AFB		15. SECURITY CLASS (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

DTIC ELECTE JAN 0 6 1988 S D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See Attached.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the DDC Ada Compiler System, Version 4.1, using Version 1.8 of the Ada® Compiler Validation Capability (ACVC). The DDC Ada Compiler System was tested on the following five configurations:

- VAX-11/785 under VMS, Release 4.3
- VAX-11/750 under VMS, Release 4.3
- MicroVAX II under MicroVMS, Release 4.4
- VAX 8200 under VMS, Release 4.4
- VAX 8650 under VMS, Release 4.4

On-site testing was performed 27 October 1986 through 31 October 1986 at DDC International in Lyngby, Denmark under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2210 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 170 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2210 tests were processed, results for Class A, C, D, or E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 31 of the processed tests determined to be inapplicable. The remaining 2179 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	99	253	332	247	161	97	136	261	128	32	217	216	2179
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	17	72	88	0	0	0	3	1	2	0	1	17	201
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

AVF Control Number: AVF-VSR-45.0587
86-09-04-DDC

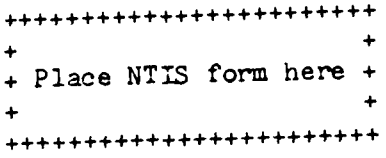
Ada® COMPILER
VALIDATION SUMMARY REPORT:
DDC International
DDC Ada Compiler System, Version 4.1
VAX-11/785, VAX-11/750, MicroVAX II,
VAX 8200, VAX 8650

Completion of On-Site Testing:
31 October 1986

Prepared By:
Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C.

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).


Place NTIS form here

Ada® Compiler Validation Summary Report:

Compiler Name: DDC Ada Compiler System, Version 4.1

Hosts and Targets:

- VAX-11/785 under VMS, Release 4.3
- VAX-11/750 under VMS, Release 4.3
- MicroVAX II under MicroVMS, Release 4.4
- VAX 8200 under VMS, Release 4.4
- VAX 8650 under VMS, Release 4.4

Testing Completed 31 October 1986 Using ACVC 1.8

This report has been reviewed and is approved.

Georgeanne C hitwood

Ada Validation Facility
Georgeanne Chitwood
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

John F Kram

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA

Virginia L Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the DDC Ada Compiler System, Version 4.1, using Version 1.8 of the Ada® Compiler Validation Capability (ACVC). The DDC Ada Compiler System was tested on the following five configurations:

- VAX-11/785 under VMS, Release 4.3
- VAX-11/750 under VMS, Release 4.3
- MicroVAX II under MicroVMS, Release 4.4
- VAX 8200 under VMS, Release 4.4
- VAX 8650 under VMS, Release 4.4

Site testing was performed 27 October 1986 through 31 October 1986 at DDC International in Lyngby, Denmark under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2210 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 170 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2210 tests were processed, results for Class A, C, D, or E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 31 of the processed tests determined to be inapplicable. The remaining 2179 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	99	253	332	247	161	97	136	261	128	32	217	216	2179
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	17	72	88	0	0	0	3	1	2	0	1	17	201
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from characteristics of particular operating systems, hardware, or implementation strategies. All of the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any unsupported language constructs required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc., under the direction of the AVF according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was conducted from 27 October 1986 through 31 October 1986 at DDC International in Lyngby, Denmark.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, FEB 1983.

2. Ada Validation Organization: Procedures and Guidelines, Ada Joint Program Office, 1 JAN 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., SEP 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. A set of programs that evaluates the conformity of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for setting procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	A test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	A test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	A test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	A test found to be incorrect and not used to check conformity to the Ada language specification. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective has been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers

permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of these units is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configurations:

Compiler: DDC Ada Compiler System, Version 4.1

ACVC Version: 1.8

Certificate Expiration Date: 17 December 1987

Host and Target Computers:

Machine	Operating System	Memory Size
VAX-11/785	VMS, Release 4.3	12 megabytes
VAX-11/750	VMS, Release 4.3	4 megabytes
MicroVAX II	MicroVMS, Release 4.4	4 megabytes
VAX 8200	VMS, Release 4.4	12 megabytes
VAX 8650	VMS, Release 4.4	12 megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. This compiler is characterized by the following interpretations of the Ada Standard:

. Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

. Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation rejects such calculations. (See tests D4A002B and D4A004B.)

. Predefined types.

This implementation supports the additional predefined types SHORT_INTEGER, LONG_INTEGER, and LONG_FLOAT in the package STANDARD. (See tests B86001C and B86001D.)

. Based literals.

An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array objects are sliced. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when

the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are not evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Functions.

An implementation may allow the declaration of a parameterless function and an enumeration literal having the same profile in the same immediate scope, or it may reject the function declaration. If it accepts the function declaration, the use of the enumeration literal's identifier denotes the function. This implementation rejects the declaration. (See test E66001D.)

. Representation clauses.

The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it. While the operation of representation clauses is not checked by Version 1.8 of the ACVC, they are used in testing other language features. This implementation accepts 'STORAGE_SIZE for collections; it rejects 'SIZE and 'SMALL clauses. Enumeration representation clauses, including those that specify noncontiguous values, appear to be supported. (See tests C55B16A, C87B62A, C87B62B, C87B62C, and BC1002A.)

. Pragmas.

The pragma INLINE is supported for procedures and for functions. (See tests CA3004E and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants. The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, AE2101H, CE2201D, CE2201E, and CE2401D.)

An existing text file can be opened and created in OUT_FILE mode and cannot be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for reading only. (See tests CE3111A..E (5 tests).)

More than one internal file can be associated with each external file for both sequential I/O and direct I/O for reading only. (See tests CE2107A..F (6 tests).)

An external file associated with more than one internal file cannot be deleted. (See test CE2110B.)

Temporary sequential and direct files are not given a name. (See tests CE2108A and CE2108C.)

. Generics.

A generic specification and body cannot be compiled in separate compilation files if the body does not come before the instantiation of the generic unit. (See tests CA2009C, CA2009F, and BC3205D.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.8 of the ACVC contains 2399 tests. When validation testing of DDC Ada Compiler System was performed, 19 tests had been withdrawn. The remaining 2380 tests were potentially applicable to this validation. The AVF determined that 201 tests were inapplicable to this implementation, and that the 2179 applicable tests were passed by the implementation.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	69	865	1171	15	13	46	2179
Failed	0	0	0	0	0	0	0
Inapplicable	0	2	197	2	0	0	201
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	99	253	332	247	161	97	136	261	128	32	217	216	2179
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	17	72	88	0	0	0	3	1	2	0	1	17	201
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

3.4 WITHDRAWN TESTS

The following 19 tests were withdrawn from ACVC Version 1.8 at the time of this validation:

C32114A	C41404A	B74101B	BC3204C
B33203C	B45116A	C87B50A	
C34018A	C48008A	C92005A	
C35904A	B49006A	C940ACA	
B37401A	B4A010C	CA3005A..D (4 tests)	

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. For this validation attempt, 201 tests were inapplicable for the reasons indicated:

- C24113I..K (3 tests) are inapplicable because they have line lengths that exceed this implementation's maximum line length.
- C34001F and C35702A use SHORT_FLOAT which is not supported by this compiler.
- D4A002B and D4A004B are inapplicable because this implementation does not support 64-bit integer calculations.

- B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

- C87B62A and C87B62C check an implementations's support of 'SIZE and 'SMALL clauses. This implementation only accepts a length clause that specifies the number of storage units to be reserved for a collection.

- C96005B checks implementations for which the smallest and largest values in type DURATION are different from the smallest and largest values in DURATION's base type. This is not the case for this implementation.

- CA2009C, CA2009F, and BC3205D compile the body and subunits of a generic unit in separate compilation files. Separate compilation of a generic specification and body is not supported by this compiler when the body comes after the instantiation of the generic unit.

- CE2102D, CE2102I and CE2111H raise USE_ERROR when an attempt is made to create a file of mode IN_FILE.

- CE2107B..E (4 tests), CE2110B, CE2111D, CE3111B..E (4 tests), and CE3114B are inapplicable because multiple internal files can be associated with the same external file for reading only. The proper exception is raised when multiple access is attempted.

- CE2108A, CE2108C, and CE3112A are inapplicable because temporary files do not have a name.

- The following 170 tests require a floating-point accuracy that exceeds the maximum of 15 supported by the implementation:

  C24113L..Y (14 tests) C35708L..Y (14 tests) C45421L..Y (14 tests)
  C35705L..Y (14 tests) C35802L..Y (14 tests) C45424L..Y (14 tests)
  C35706L..Y (14 tests) C45241L..Y (14 tests) C45521L..Z (15 tests)
  C35707L..Y (14 tests) C45321L..Y (14 tests) C45621L..Z (15 tests)

3.6 SPLIT TESTS

If one or more errors do not appear to have been detected in a Class B test because of compiler error recovery, then the test is split into a set of smaller tests that contain the undetected errors. These splits are then compiled and examined. The splitting process continues until all errors are detected by the compiler or until there is exactly one error per split. Any Class A, Class C, or Class E test that cannot be compiled and executed because of its size is split into a set of smaller subtests that can be processed.

Splits were required for seven Class B tests.

B33301A	B67001A	BA1101B
B37302A	B67001C	
B55A01A	B67001D	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.8 produced by the DDC Ada Compiler System was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the DDC Ada Compiler System using ACVC Version 1.8 was conducted on-site by a validation team from the AVF. The configuration consisted of a VAX-11/785 operating under VMS, Release 4.3. The following four configurations were also tested using a subset of the ACVC:

- VAX-11/750 under VMS, Release 4.3
- MicroVAX II under MicroVMS, Release 4.4
- VAX 8200 under VMS, Release 4.4
- VAX 8650 under VMS, Release 4.4

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring splits during the prevalidation testing were included in their split form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the VAX-11/785. After the test files were loaded to disk, the full set of tests was compiled on the VAX-11/785, and all executable tests were linked and run. Results were printed from the VAX-11/785. The tests were reviewed by the validation team and showed acceptable results.

A subset of the ACVC, Version 1.8, was run on a VAX-11/750, a MicroVAX II, a VAX 8200, and a VAX 8650. The subset of sixty tests consisted of five tests selected at random from all classes of tests within each chapter. The tests were compiled, linked, and executed as appropriate. The test

results were the same as those reviewed for the VAX-11/785 on which full testing was performed.

The compiler was tested on both computers using command scripts provided by DDC International and reviewed by the validation team. The following options were in effect for testing:

Option	Effect
/LIST	List file is created during compilation.

Test output, compilation listings, job logs, and the compiler and environment were written to magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

The validation team arrived at DDC International in Lyngby, Denmark on 27 October 1986, and departed after testing was completed on 31 October 1986.

APPENDIX A

COMPLIANCE STATEMENT

DDC International has submitted the following compliance statement concerning the DDC Ada Compiler System.

COMPLIANCE STATEMENT

Compliance Statement

Configuration:

Compiler: DDC Ada®Compiler System, Version 4.1

Test Suite: Ada Compiler Validation Capability, Version 1.8

Host and Target Computers:

Machine:	VAX_11/785
Operating System:	VMS, Release 4.3
Machine:	VAX_11/750
Operating System:	VMS, Release 4.3
Machine:	VAX_8650
Operating System:	VMS, Release 4.4
Machine:	MicroVAX II
Operating System:	VMS, Release 4.4
Machine:	VAX 8200
Operating System:	VMS, Release 4.4

DDC International has made no deliberate extensions to the Ada language standard.

DDC International agrees to the public disclosure of this report.

DDC International agrees to comply with the Ada trademark policy, as defined by the Ada Joint Program Office.

Carsten Bjernaa Date: 31/10-96

DDC International
Carsten Bjernaa
Project Manager

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation classes. The implementation-dependent characteristics of the DDC Ada Compiler System, Version 4.1, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

     ...

     type INTEGER is range -32768 .. 32767;
     type SHORT_INTEGER is range -128 .. 127;
     type LONG_INTEGER is range -2147483648 .. 2147483647;

     type FLOAT is digits 6 range -16#7.FFFF_C#E31 .. 16#7.FFFF_C#E31;
     type LONG_FLOAT is digits 15 range -16#7.FFFF_FFFF_FFFF#E255 ..
          16#7.FFFF_FFFF_FFFF#E255;

     type DURATION is delta 2#1.0#E-14 range -131072.0 .. 131071.0;
       -- DURATION'SMALL = 2#1.0#E-14.

     ...

end STANDARD;
```

F. Appendix F of the Ada Reference Manual

F.0 Introduction

This appendix describes the implementation-dependent characteristics of the DDC VAX/VMS Ada Compiler, as required in the Appendix F frame of the Ada Reference Manual (ANSI/MIL-STD-1815A).

F.1 Implementation-Dependent Pragmas

No implementation-dependent pragmas are defined for the VAX/VMS version.

F.2 Implementation-Dependent Attributes

No implementation-dependent attributes are defined for the VAX/VMS version.

F.3 Package SYSTEM

The specification of the package SYSTEM:

```
package SYSTEM is

   type ADDRESS          is access INTEGER;
   subtype PRIORITY      is INTEGER range 0..15;
   type NAME             is (VAX11,CR80,M40,MPS10,DPS6);
   SYSTEM_NAME:          constant NAME  := VAX11;
   STORAGE_UNIT:         constant       := 16;
   MEMORY_SIZE:          constant       := 2048 * 1024;
   MIN_INT:              constant       := -2_147_483_647-1;
   MAX_INT:              constant       := 2_147_483_647;
   MAX_DIGITS:           constant       := 15;
   MAX_MANTISSA:         constant       := 31;

   FINE_DELTA:           constant       := 2.0 / MAX_INT;
   TICK:                 constant       := 0.000_001;

end SYSTEM;
```

F.4 Representative Clauses

In general, no representation clauses may be given for a derived type. The representation clauses that are accepted for non-derived types are described in the following:

Length Clause

The compiler accepts only a length clause that specifies the number of storage units to be reserved for a collection.

Enumeration Representation Clause

Enumeration representation clauses may specify representations only in the range of the predefined type INTEGER.

Record Representation Clause

A component clause is allowed if and only if

- the component type is a discrete type different from LONG_INTEGER
- the component type is an array type with a discrete element type different from LONG_INTEGER.

No component clause is allowed if the component type is not covered by the above two inclusions. If the record type contains components not covered by a component clause, they are allocated consecutively after the component with the value. Allocation of a record component without a component clause is always aligned on a storage unit boundary. Holes created because of component clauses are not otherwise utilized by the compiler.

F.5 Implementation-Dependent Names for Implementation - Dependent Components

None defined by the compiler.

F.6 Address Clauses

Not supported by the compiler.

F.7 Unchecked Conversion

Unchecked conversion is only allowed between values of the same "size". In this context the "size" of an array is qual to that of two access values and the "size" of a packed array is equal to two access values and an integer. This is the only restriction imposed on unchecked conversion.

F.8 Input-Output Packages

The implementation supports all requirements of the Ada language. It is an effective interface to the VAX/VMS file system, and in case of text input-output also an effective interface to the VAX/VMS terminal driver.

This section describes the functional aspects of the interface to the VAX/VMS file system and terminal driver. Certain portions of this section is of special interest to the system programmer who needs to control VAX/VMS specific Input-Output characteristics via Ada programs.

The section is organised as follows.

Subsection numbers refer to the equivalent subsections in Chapter 14 of the ARM. Only subsections of interest to this section are included.

The Ada Input-Output concept as defined in Chapter 14 of the ARM does not constitute a complete functional specifications of the Input-Output packages. Some aspects are not discussed at all, while others are deliberately left open to an implementation.

These gaps are filled in the appropriate subsections and summarized in subsection F.8.a.

The reader should be familiar with

[DoD 83] - The Ada language definition

and certain sections require that the reader is familiar with

[DEC 84a] - Guide to VAX/VMS File Applications

[DEC 84b] - Record Management Services

[DEC 85] - VAX/VMS I/O Users Reference Manual

F.8.1 External Files and File Objects

An external file is either any VAX/VMS file residing on a file-structured device (disk, tape), a record structured device (terminal, lineprinter), or a virtual software device (mailbox). ARM 14.1(1).

Identification of an external file by a string (the NAME parameter) is described in subsection F.8.2.1.

System-dependent characteristics (the FORM parameter) is described in subsection F.8.2.1

An external file created on a file-structured device will exist after program termination, and may be accessed later from an Ada program, except if the file is a temporary file created by using an empty name parameter. If files corresponding to the external file have not been closed, the external file will also exist upon program completion, and the contents will be the same as if the files had been closed prior to program completion. See further F.8.3. ARM 14.1(7).

Input-Output of access types will cause input-output of the access value [Dod 83] 14.1(7).

Sharing of an external file is, when using the default system-dependent characteristics, handled as described in the following.

When a file is associated with an external file using the Record Management Services (RMS), and the file is opened with mode IN_FILE, the implementation will allow the current process and other processes to open files associated with the same external file (e.g. as IN_FILE in an Ada program).

When a file is opened with mode INOUT_FILE or OUT_FILE no file sharing is allowed when using RMS. In particular, trying to gain write access to an external file shared by other files, by OPEN or RESET to mode INOUT_FILE or OUT_FILE will raise USE_ERROR.

When a text file is associated with a terminal device, using the Queue I/O System Services (QIO), there are no restrictions on file sharing.

F.8.2 Sequential and Direct Files

When dealing with sequential and direct input-output only RMS files are used.

In this section, a description of the basic file-mapping is given.

Basic file-mapping concerns the relation between Ada files and (formats of) external RMS files, and the strategy for accessing the external files. When creating new files (with the CREATE procedure), there is a unique mapping onto a RMS file format, the preferred file format. When opening an existing external file (with the OPEN procedure), the mapping is not unique; i.e. several external file formats other than preferred for CREATE may be acceptable. In subsection F.8.2.1 the preferred and acceptable formats are described for sequential and direct input-output. In subsection F.8.3.1 the preferred and acceptable formats are described for text input-output.

F.8.2.1 File Management

This subsection contains information regarding file management:

- Description of preferred and acceptable formats for sequential and direct input-output.
- The NAME parameter.
- The FORM parameter.
- File access.

Preferred and Acceptable Formats

The preferred and acceptable formats for sequential and direct input-output, are described using RMS notation and abbreviations [DEC 84b]. ES is used to denote the element size, i.e. the number of bytes occupied by the element type, or, in case of a varying size type, the maximum size (which must be determinable at the point of instantiation from the value of the SIZE attribute for the element type).

It should be noted that the latter means a type definition like:

```
type large_type is array( integer <> ) of integer;
```

would be mapped onto an element size greater than the maximum allowed size (32 k byte).

SEQUENTIAL_IO:

An element is mapped into a single record of the external file, or if block-io is used, a number of consecutive virtual blocks of 512 bytes. ES must not be greater than 32767, otherwise USE_ERROR is raised.

CREATE - preferred file format

- ORG=SEQ, RFM=FIX, MRS=ES
  (note: read and write operations will be done by BLOCK IO if element size is a multiple of 512 bytes)

OPEN - acceptable formats

- ORG=REL, RFM=FIX, MRS=ES
- ORG=SEQ, RFM=FIX, MRS=ES

- ORG=SEQ, RFM=VAR

- ORG=SEQ, RFM=UDF
  (note: BLOCK IO will be used)

  (note: a RESET operation to OUT_FILE mode will give a USE_ERROR exception, as it is not possible to empty a file of this format).

The detailed setting of the control blocks for sequential_IO is given below. Note that the user-provided form parameter will override the default specified settings, when used with OPEN or CREATE.

Also note that, when an Ada program contains tasks, asynchronous I/O will be used (ROP = <ASY>).

The following shows the initial setting for OPEN and CREATE (unspecified fields in the control blocks will be cleared to zero).

```
FAB:
  ALQ = 12
  DEQ = 6
  DNM = <.DAT>
  FAC = for block-io, IN_FILE:     <BRO,GET>
        for block-io, OUT_FILE:    <BRO,PUT,UPD,DEL,TRN>
        otherwise,    IN_FILE:     <GET>
        otherwise,    OUT_FILE:    <PUT,UPD,DEL,TRN>
  FNM = name parameter
  FOP = non-empty name parameter:          <MXV,SQO>
        empty name parameter to CREATE: <MXV,SQO,TMP>
  MRS = element size (in bytes)
  NAM = address of name-block
  ORG = SEQ
  RAT = <CR>
  RFM = FIX
  SHR = for IN_FILE:   <GET>
        for OUT_FILE:  <NIL>
  XAB = address of XABFHC block

RAB:
  FAB = address of FAB block
  KBF = address of internal longword
  KSZ = 4
  RAC = SEQ
  ROP = for block-io:  <BIO>
        otherwise:     <UIF>
NAM:
  RSA = address of internal 255 byte buffer
  RSS = 255
```

```
  XABFHC:
    NXT = 0
DIRECT_IO:
```

An element is mapped into a single record of the external file, or if block io is used, the smallest possible number of consecutive virtual blocks of 512 bytes. ES must not be greater than 32767, otherwise USE_ERROR will be raised.

CREATE - preferred file format

- if element size is not a multiple of 512:
  ORG=REL, RFM=FIX, MRS=ES

- if element size is a multiple of 512: ORG=SEQ, REM=FIX, MRS=ES
  (note: read and write operations will be done by BLOCK IO)

OPEN - acceptable formats

- ORG=REL, RFM=FIX, MRS=ES

- ORG=SEQ, RFM=FIX, MRS=ES
  (note: if element size is a multiple of 512, BLOCK IO will be used)

- ORG=SEQ, RFM=UDF
  (note: BLOCK IO will be used)

The detailed setting of the control blocks for direct_IO is given below. Note that the user-provided form parameter will override the default specified settings, when used with OPEN or CREATE.

Also note that, when an Ada program contains tasks, asynchronous I/O will be used (ROP = <ASY>).

The initial setting for OPEN and CREATE (unspecified fields in the control blocks will be cleared to zero) follows:

```
  FAB:
    ALQ = 12
    DEQ = 6
    DNM = <.DAT>
    FAC = for IN_FILE:  <GET>
          for OUT_FILE: <GET,PUT,UPD,DEL,TRN>
    FNM = name parameter
```

```
    FOP = non-empty name parameter:        <MXV,SQO>
          empty name parameter to CREATE: <MXV,SQO,TMP>
    MRS = 512
    NAM = address of name-block
    ORG = SEQ
    RAT = <CR>
    RFM = VAR
    SHR = for IN_FILE:  <GET>
          for OUT_FILE: <NIL>
    XAB = address of XABFHC block

  RAB:
    FAB = address of FAB block
    KBF = address of internal longword
    KSZ = 4
    RAC = SEQ
    ROP = <>
    UBF = address of internal 512 byte buffer
    USZ = 512

  NAM:
    RSA = address of internal 255 byte buffer
    RSS = 255

  XABFHC:
    NXT = 0
```

Name Parameter

The name parameter, when non null, must be a valid VAX/VMS file specification referring to a file-structured device; a file with that name will then be created.

For a null name parameter, the process' current directory and device must designate a directory on a disk device; a temporary, unnamed file marked for deletion will then be created in that directory. The file will be deleted after closing it, or, if not closed when the program terminates. ARM 14.2.1(3).

Form Parameter

The FORM string parameter that can be supplied to any OPEN or CREATE procedure is for controlling the external file properties, such as physical organization, allocation etc. In the present implementation this has been achieved by accepting form parameters that specify setting of fields in the RMS control blocks FAB and RAB, used for all open files. This scheme is rather general in that it accepts all settings of the FAB and RAB fields. It opens for modifications of the behaviour required by the Arm, such as being able to open a file for

appending data to it. Furthermore, a form parameter for accessing mailboxes is provided.

The following fields can currently not be set explicitly:

```
FAB:
  FNA, FNS (are set by the NAME parameter of OPEN or
            CREATE)
  DNA, DNS (can be set by DNM=/.../)
```

The syntax of the form parameter is as follows:

```
form_parameter ::= [ param { , param } ]

param          ::= number_param
                 | string_param
                 | quotation_param
                 | mask_param

number_param   ::= keyword = number
number         ::= digit { digit }
digit          ::= 0 | 1 | ... | 9
string_param   ::= keyword = string
string         ::= / {any character other than slash} /

quotation_param::= keyword = specifier

mask_param     ::= clear_bits
                 | set_bits
                 | define_whole_field

clear_bits     ::= keyword - mask
set_bits       ::= keyword + mask
define_whole_field
               ::= keyword = mask
mask           ::= < [ specifier { , specifier } ] >

keyword        ::= letter letter letter
specifier      ::= letter letter letter [ letter letter ]

letter         ::= A | B | ... | Z | a | b | ... | z
```

Notes:

- all space characters are ignored.
- string parameters are converted to uppercase.
- all keywords and specifiers are 3- or 5-letter words, like the RMS assembly level interface symbolic names. The only exceptions are the RAT=<CR> specifier, which in this implementation must be specified as CAR rather than CR, and the RAB CTX field keyword, which must be

specified as CON. There are only 2 5-letter words: the specifiers STMCR and STMLF.

The semantics of the form parameter is (except for the mailbox parameter) to modify the specified FAB and RAB fields just prior to actually calling RMS to open or create a file, i.e. the form parameter overrides the default conventions provided by this implementation (ARM section F.5.4). The form parameter is interpreted left to right, and it is legal to respecify fields; in particular a mask field may be manipulated in several turns.

Note that there is no way of modifying fields after an RMS open or create service, in particular it is not possible to set RAB fields on a per record operation basis.

The modifications made are those to be expected from the textually corresponding RMS macro specifications. However, the clear_bits and set_bits are particular to this implementation: They serve to either clear individual mask specifiers set by the implementation default, or to set mask specifiers in addition to those specified by the implementation default, respectively.
The mailbox parameter can be either

```
MBX=TMP
```

or

```
MBX=PRM
```

It applies to CREATE only, and causes either a temporary or a permanent mailbox to be created. The NAME parameter will be used to establish a logical name for the mailbox, unless an empty string is specified (in this case, no logical name will be established).

Note that the implementation does in no way check that the form parameter supplied is at all reasonable. The attitude is "you asked for it, you got it". It is discouraged, if other procedures than OPEN, CREATE, and CLOSE will be called, to set ORG, RAC, MRS, NAM, FOP=<NAM>. It is generally discouraged to set XAB.

Examples:

```
-- create a text file
create(file, out_file, "DATA.TXT");

-- create a temporary text file which will be deleted
      after completion of the main program
create(file, out_file);

-- create an empty stream format text file
create(file,out_file,"DATA.DAT","ORG=SEQ,RFM=STMLF");

-- create a very big file:
create(file,out_file,"DATA.DAT","ALQ=2048,DEQ=256");

-- create a temporary mailbox:
create(file,out_file,"HELLO","MBX=TMP");

-- open a mailbox; at reading, do not wait for
      messages:
open(file,in_file,"HELLO","ROP+<TMO>,TMO=0");
```

File Access

The OPEN and CREATE procedures utilize the normal RMS defaulting mechanism to determine the exact file to open or create.

Device and directory (when not specified) defaults to the process' current device (SYS$DISK) and directory.

The version number (when not specified), defaults for OPEN to highest existing, or for CREATE, one higher than the highest existing, or 1 when no version exists.

The implementation provides .DAT as the default file type.

External files, which are not to be accessed via block-io (as described in formats), will be accessed via standard RMS access methods. For SEQUENTIAL_IO, sequential record access mode will be used. For DIRECT_IO, random access by record number will be used.

Creation of a file with mode IN_FILE will raise USE_ERROR, when referring to an RMS file.

For sequential and direct io, files created by SEQUENTIAL_IO for a given type T, may be opened (and processed) by DIRECT_IO for the same type and vice-versa. In the latter case, however, the function END_OF_FILE (14.2.2(8)) may fail to produce TRUE in case where the file has been written at random, leaving "holes" in the file. See ARM 14.2.1(7).

For a sequential or text file associated with an RMS file, a RESET operation to OUT_FILE mode will cause deletion of any elements in the file, i.e. the file is emptied. Likewise, a sequential file or text file opened (by OPEN) with mode OUT_FILE, will be emptied. For any other RESET operation, the contents of the file is not affected.

For a text file, any RESET operation will cause USE_ERROR to be raised, when QIO services are used.

F.8.2.2 Sequential Input-Output

The implementation omits type checking for DATA_ERROR, in case the element type is of an unconstrained type, ARM 14.2.2(4), i.e.:

```
... f : FILE_TYPE
type et  is 1..100;
type eat is array( et range <> ) of integer;

X : eat( 1..2 );
Y : eat( 1..4 );
...
-- write X, Y:

write( f, X); write( f, Y); reset( f, IN_FILE);

-- read X into Y and Y into X:

read( f, Y); read( f, X);
```

This should have given DATA_ERROR, but will instead give undefined values in the last 2 elements of Y.

F.8.2.3 Specification of the Package Sequential_IO

```
with BASIC_IO_TYPES;

with IO_EXCEPTIONS;

generic

   type ELEMENT_TYPE is private;

package SEQUENTIAL_IO is

  type FILE_TYPE is limited private;

  type FILE_MODE is (IN_FILE, OUT_FILE);
```

```
-- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE := OUT_FILE;
                    NAME : in     STRING    := "";
                    FORM : in     STRING    := "");

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE;
                    NAME : in     STRING;
                    FORM : in     STRING := "");

   procedure CLOSE (FILE : in out FILE_TYPE);

   procedure DELETE(FILE : in out FILE_TYPE);

   procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE);

   procedure RESET (FILE : in out FILE_TYPE);

   function MODE   (FILE : in FILE_TYPE) return FILE_MODE;

   function NAME   (FILE : in FILE_TYPE) return STRING;

   function FORM   (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

-- input and output operations

   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE);

   procedure WRITE (FILE : in FILE_TYPE;
                    ITEM : in ELEMENT_TYPE);

   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

-- exceptions

   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

private

   type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end SEQUENTIAL_IO;
```

F.8.2.4 Direct Input-Output

The implementation omits type checking for DATA_ERROR, in case the element type is of an unconstrained type, [Dod 83] 14.2.4(4), see F.8.2.2.

F.8.2.5 Specification of the Package Direct IO

```
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;

generic

   type ELEMENT_TYPE is private;

package DIRECT_IO is


   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);

   type COUNT is range 0..LONG_INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;


-- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE  := INOUT_FILE;
                    NAME : in     STRING     := "";
                    FORM : in     STRING     := "");

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE;
                    NAME : in     STRING;
                    FORM : in     STRING    := "");

   procedure CLOSE (FILE : in out FILE_TYPE);

   procedure DELETE(FILE : in out FILE_TYPE);

   procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE);

   procedure RESET (FILE : in out FILE_TYPE);

   function MODE   (FILE : in FILE_TYPE) return FILE_MODE;

   function NAME   (FILE : in FILE_TYPE) return STRING;
```

```
   function FORM   (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

-- input and output operations

   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE;
                    FROM : in     POSITIVE_COUNT);
   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE);

   procedure WRITE (FILE : in FILE_TYPE;
                    ITEM : in ELEMENT_TYPE;
                    TO   : in POSITIVE_COUNT);
   procedure WRITE (FILE : in FILE_TYPE;
                    ITEM : in ELEMENT_TYPE);

   procedure SET_INDEX(FILE : in FILE_TYPE;
                       TO   : in POSITIVE_COUNT);

   function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;

   function SIZE (FILE : in FILE_TYPE) return COUNT;

   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;


-- exceptions

   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

private

  type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end DIRECT_IO;
```

F.8.3 Text Input-Output

When utilizing text input-output, RMS is used when an external file is residing on a file-structured device, or is a virtual software device. When an external file that is a terminal device is opened or created, the queue I/O services (QIO) are used by default.

If a text file of mode OUT_FILE corresponds to an external RMS file, the external file will also exist upon program completion, and a pending linebuffer will be flushed before the text file is closed.

F.8.3.1 File management

This subsection contains information regarding file management, where it differs from the file management described in F.8.2.1.

- Description of preferred and acceptable formats for text input-output.
- The FORM parameter.
- File access.

Preferred and Acceptable Formats

Lines of text are mapped into records of external files.

For output, the following rules apply.

The Ada line terminators and file terminators are never explicitly stored (however, for stream format files, RMS forces line terminators to trail each record). Page terminators, except the last, are mapped into a form feed character trailing the last line of the page. (In particular, an empty page (except the last) is mapped into a single record containing only a form feed character). The last page terminator in a file is never represented in the external file. It is not possible to write records containing more than 512 characters. That is, the maximum line length is 511 or 512, depending on whether a page terminator (form feed character) must be written or not. If output is more than 512 characters, USE_ERROR will be raised.

On input, a FF trailing a record indicates that the record contains the last line of a page and that at least one more page exists. The physical end of file indicates the end of the last page.

CREATE - preferred file format

- ORG=SEQ, RFM=VAR, MRS=512

OPEN - acceptable file formats

- all formats except
  - ORG=IDX
  - RFM=UDF

(Note: for stream files (RFM=STM...) any sequence of the LF, CR, and VT control characters at the end of a line will be stripped off at input. At output, line terminators will be provided by RMS defaults). (Note: input of any record containing more than 512 characters will raise a USE_ERROR exception).

The detailed setting of the control blocks for TEXT_IO is given below. Note that the user-provided form parameter will override the default specified settings, when used with OPEN or CREATE.

Also note that, when an Ada program contains tasks, asynchronous I/O will be used. When RMS files ROP = <ASY>, or asynchronous QIO when terminal devices.

The following shows the initial setting for OPEN and CREATE (unspecified fields in the control blocks will be cleared to zero):

```
FAB:
  ALQ = 12
  DEQ = 6
  DNM = <.DAT>
  FAC = for IN_FILE:      <GET>
        for OUT_FILE:     <GET,PUT,UPD,DEL,TRN>
  FNM = name parameter
  FOP = non-empty name parameter              <MXV,SQO>
        empty name parameter to CREATE:       <MXV,SQO,TMP>
  MRS = 512
  NAM = address of name-block
  ORG = SEQ
  RAT = <CR>
  RFM = VAR
  SHR = for IN_FILE:     <GET>
        for OUT_FILE:    <NIL>
  XAB = address of XABFHC block

RAB:
  FAB = address of FAB block
  KBF = address of internal longword
  KSZ = 4
  RAC = SEQ
  ROP = <>
  UBF = address of internal 512 byte buffer
  USZ = 512
```

```
NAM:
  RSA = address of internal 255 byte buffer
  USZ = 255

XABFHC:
  NXT = 0
```

Form parameter

If any form parameter, except for the empty string or a string containing only blanks, is supplied to OPEN or CREATE, RMS services will always be used. In this case, the file operations on external files as terminal-devices will use buffered input- output.

File access

External RMS files are accessed via sequential record access methods.

Files associated with terminal devices, using QIO services, do not contain page terminators. This means that calling SKIP_PAGE will raise USE_ERROR. Furthermore, trying to RESET a file in this category will cause USE_ERROR.

Files associated with the same external file, using QIO services, share the standard values (page-, line, and column-number), e.g. standard values for STANDARD_OUTPUT are implicitly updated after reading from STANDARD_INPUT.

F.8.3.10 Specification of the Package Text IO

```
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;
package TEXT_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, OUT_FILE);

   type    COUNT is range 0 .. LONG_INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
   UNBOUNDED: constant COUNT:= 0; -- line and page length

   subtype FIELD          is INTEGER range 0 .. 35;

   subtype NUMBER_BASE    is INTEGER range 2 .. 16;

   type TYPE_SET is (LOWER_CASE, UPPER_CASE);
```

```
-- File Management

  procedure CREATE (FILE   : in out FILE_TYPE;
                    MODE   : in     FILE_MODE := OUT_FILE;
                    NAME   : in     STRING    := "";
                    FORM   : in     STRING    := ""
                   );

  procedure OPEN   (FILE   : in out FILE_TYPE;
                    MODE   : in     FILE_MODE;
                    NAME   : in     STRING;
                    FORM   : in     STRING    := ""
                   );

procedure CLOSE  (FILE   : in out FILE_TYPE);
procedure DELETE (FILE   : in out FILE_TYPE);
procedure RESET  (FILE   : in out FILE_TYPE;
                  MODE   : in     FILE_MODE);
procedure RESET  (FILE   : in out FILE_TYPE);

function  MODE   (FILE   : in FILE_TYPE) return FILE_MODE;
function  NAME   (FILE   : in FILE_TYPE) return STRING;
function  FORM   (FILE   : in FILE_TYPE) return STRING;

function  IS_OPEN(FILE   : in FILE_TYPE) return BOOLEAN;

-- Control of default input and output files

procedure  SET_INPUT   (FILE : in FILE_TYPE);
procedure  SET_OUTPUT  (FILE : in FILE_TYPE);

function   STANDARD_INPUT   return FILE_TYPE;
function   STANDARD_OUTPUT  return FILE_TYPE;

function   CURRENT_INPUT    return FILE_TYPE;
function   CURRENT_OUTPUT   return FILE_TYPE;


-- specification of line and page lengths

procedure SET_LINE_LENGTH  (FILE : in FILE_TYPE;
                            TO   : in COUNT);
procedure SET_LINE_LENGTH  (TO   : in COUNT);

procedure SET_PAGE_LENGTH  (FILE : in FILE_TYPE;
                            TO   : in COUNT);
procedure SET_PAGE_LENGTH  (TO   : in COUNT);

function  LINE_LENGTH      (FILE : in FILE_TYPE) return
                                                 COUNT;
function  LINE_LENGTH                            return
                                                 COUNT;
```

```
   function  PAGE_LENGTH      (FILE : in FILE_TYPE) return
                                                    COUNT;
   function  PAGE_LENGTH                            return
                                                    COUNT;

-- Column, Line, and Page Control

   procedure NEW_LINE    (FILE    : in FILE_TYPE;
                          SPACING : in POSITIVE_COUNT := 1);
   procedure NEW_LINE    (SPACING : in POSITIVE_COUNT := 1);

   procedure SKIP_LINE   (FILE    : in FILE_TYPE;
                          SPACING : in POSITIVE_COUNT := 1);
   procedure SKIP_LINE   (SPACING : in POSITIVE_COUNT := 1);

   function  END_OF_LINE (FILE : in FILE_TYPE) return
                                                BOOLEAN;
   function  END_OF_LINE                        return
                                                BOOLEAN;

   procedure NEW_PAGE    (FILE : in FILE_TYPE);
   procedure NEW_PAGE                         ;

   procedure SKIP_PAGE   (FILE : in FILE_TYPE);
   procedure SKIP_PAGE                        ;

   function  END_OF_PAGE (FILE : in FILE_TYPE) return
                                                BOOLEAN;
   function  END_OF_PAGE                        return
                                                BOOLEAN;

   function  END_OF_FILE (FILE : in FILE_TYPE) return
                                                BOOLEAN;
   function  END_OF_FILE                        return
                                                BOOLEAN;

   procedure SET_COL     (FILE : in FILE_TYPE;
                          TO   : in POSITIVE_COUNT);
   procedure SET_COL     (TO   : in POSITIVE_COUNT);

   procedure SET_LINE    (FILE : in FILE_TYPE;
                          TO   : in POSITIVE_COUNT);
   procedure SET_LINE    (TO   : in POSITIVE_COUNT);

   function  COL         (FILE : in FILE_TYPE) return
                                      POSITIVE_COUNT;
   function  COL                                return
                                      POSITIVE_COUNT;

   function  LINE        (FILE : in FILE_TYPE) return
                                      POSITIVE_COUNT;
   function  LINE                               return
                                      POSITIVE_COUNT;
```

```
   function  PAGE        (FILE : in FILE_TYPE) return
                                    POSITIVE_COUNT;
   function  PAGE                                 return
                                    POSITIVE_COUNT;

 -- Character Input-Output

 procedure GET  (FILE : in     FILE_TYPE;
                 ITEM :    out CHARACTER);
 procedure GET  (ITEM :    out CHARACTER);
 procedure PUT  (FILE : in FILE_TYPE;
                 ITEM : in CHARACTER);
 procedure PUT  (ITEM : in CHARACTER);

 -- String Input-Output

 procedure GET  (FILE : in     FILE_TYPE;
                 ITEM :    out STRING);
 procedure GET  (ITEM :    out STRING);
 procedure PUT  (FILE : in FILE_TYPE;
                 ITEM : in STRING);
 procedure PUT  (ITEM : in STRING);

 procedure GET_LINE  (FILE : in     FILE_TYPE;
                      ITEM :    out STRING;
                      LAST :    out NATURAL);
 procedure GET_LINE  (ITEM :    out STRING;
                      LAST :    out NATURAL);
 procedure PUT_LINE  (FILE : in     FILE_TYPE;
                      ITEM : in     STRING);
 procedure PUT_LINE  (ITEM : in     STRING);


 -- Generic Package for Input-Output of Integer Types

generic
    type NUM is range <>;
package INTEGER_IO is

   DEFAULT_WIDTH : FIELD        := NUM'WIDTH;
   DEFAULT_BASE  : NUMBER_BASE  :=        10;

   procedure GET  (FILE  : in     FILE_TYPE;
                   ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);
   procedure GET  (ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);
```

```
      procedure PUT  (FILE  : in FILE_TYPE;
                      ITEM  : in NUM;
                      WIDTH : in FIELD := DEFAULT_WIDTH;
                      BASE  : in NUMBER_BASE := DEFAULT_BASE);
      procedure PUT  (ITEM  : in NUM;
                      WIDTH : in FIELD := DEFAULT_WIDTH;
                      BASE  : in NUMBER_BASE := DEFAULT_BASE);

      procedure GET  (FROM  : in     STRING;
                      ITEM  :    out NUM;
                      LAST  :    out POSITIVE);
      procedure PUT  (TO    :    out STRING;
                      ITEM  : in     NUM;
                      BASE  : in     NUMBER_BASE :=
                                          DEFAULT_BASE);

   end INTEGER_IO;


    -- Generic Packages for Input-Output of Real Types

generic
   type NUM is digits <>;
package FLOAT_IO is

   DEFAULT_FORE : FIELD :=              2;
   DEFAULT_AFT  : FIELD := NUM'digits - 1;
   DEFAULT_EXP  : FIELD :=              3;

   procedure GET  (FILE  : in     FILE_TYPE;
                   ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);
   procedure GET  (ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);

   procedure PUT  (FILE : in FILE_TYPE;
                   ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);
   procedure PUT  (ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure GET  (FROM : in     STRING;
                   ITEM :    out NUM;
                   LAST :    out POSITIVE);
   procedure PUT  (TO   :    out STRING;
                   ITEM : in     NUM;
                   AFT  : in     FIELD := DEFAULT_AFT;
                   EXP  : in     FIELD := DEFAULT_EXP);

end FLOAT_IO;
```

```
generic
   type NUM is delta <>;
package FIXED_IO is

   DEFAULT_FORE : FIELD := NUM'FORE;
   DEFAULT_AFT  : FIELD := NUM'AFT;
   DEFAULT_EXP  : FIELD := 0;

   procedure GET  (FILE  : in     FILE_TYPE;
                   ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);
   procedure GET  (ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);

   procedure PUT  (FILE : in FILE_TYPE;
                   ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure PUT  (ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure GET  (FROM : in     STRING;
                   ITEM :    out NUM;
                   LAST :    out POSITIVE);
   procedure PUT  (TO   :    out STRING;
                   ITEM : in     NUM;
                   AFT  : in     FIELD := DEFAULT_AFT;
                   EXP  : in     FIELD := DEFAULT_EXP);

end FIXED_IO;


   -- Generic Package for Input-Output of Enumeration Types

generic
   type ENUM is (<>);
package ENUMERATION_IO is

   DEFAULT_WIDTH   : FIELD    := 0;
   DEFAULT_SETTING : TYPE_SET := UPPER_CASE;

   procedure GET  (FILE : in     FILE_TYPE;
                   ITEM :    out ENUM);
   procedure GET  (ITEM :    out ENUM);

   procedure PUT  (FILE  : in FILE_TYPE;
                   ITEM  : in ENUM;
                   WIDTH : in FIELD      := DEFAULT_WIDTH;
                   SET   : in TYPE_SET   := DEFAULT_SETTING);
```

```
    procedure PUT  (ITEM  : in ENUM;
                    WIDTH : in FIELD      := DEFAULT_WIDTH;
                    SET   : in TYPE_SET   := DEFAULT_SETTING);

    procedure GET  (FROM : in     STRING;
                    ITEM :    out ENUM;
                    LAST :    out POSITIVE);
    procedure PUT  (TO   :    out STRING;
                    ITEM : in     ENUM;
                    SET  : in     TYPE_SET := DEFAULT_SETTING);

end ENUMERATION_IO;

    -- Exceptions

    STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
    MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
    NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
    DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
    END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
    DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
    LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

private

  type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end TEXT_IO;
```

F.8.6 Low Level Input-Output

The package LOW_LEVEL_IO is empty.

F.8.a Clarifications of Ada Input-Output Requirements Summary

The Ada Input-Output concepts as presented in Chapter 14 of ARM do not constitute a complete functional specification of the Input-Output packages. Some aspects are not discussed at all, while others are deliberately left open to an implementa- tion. These gaps are filled in below, with reference to sections of the ARM.

F.8.b Assumptions

14.2.1(15): For a sequential or text file, a RESET operation to OUT_FILE mode deletes any elements in the file, i.e. the file is emptied. Likewise, a sequential or text file opened (by OPEN) as an OUT_FILE, will

be emptied. For any other RESET operation, the contents of the file is not affected.

14.2.1(7) : For sequential and direct io, files created by SEQUENTIAL_IO for a given type T, may be opened (and processed) by DIRECT_IO for the same type and vice-versa. In the latter case, however, the function END_OF_FILE (14.2.2(8)) may fail to produce TRUE in the case where the file has been written at random, leaving "holes" in the file.

F.8.c Implementation Choices

14.1(1) : An external file is either any VAX/VMS file residing on a file-structured device (disk,tape), a record structured device (terminal, lineprinter), or a virtual software device (mailbox).

14.1(7) : An external file created on a file-structured device will exist after program termination, and may later be accessed from an Ada program.

14.1(13) : See Section F.8.2.1 File Management.

14.2.1(3) : The name parameter, when non-null, must be a valid VAX/VMS file specification referring to a file-structured device; a file with that name will then be created. For a null name parameter, the process' current directory and device must designate a directory on a disk device; a temporary, unnamed file marked for deletion will then be created in that directory.

The form and effect of the form parameter is discussed in Sections F.8.2.1 and F.8.3.1.

Creation of a file with mode IN_FILE will raise USE_ERROR.

14.2.1(13): Deletion of a file is only supported for files on a disk device, and requires deletion access right to the file.

14.2.2(4): No check for DATA_ERROR is performed in case the element type is of an unconstrained type.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..125 => 'A', 126 => '1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..125 => 'A', 126 => '2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..63 => 'A', 64 => '3', 65..126 => 'A')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..63 => 'A', 64 => '4', 65..126 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..123 => '0', 124..126 => "298")

Name and Meaning	Value
$BIG_REAL_LIT A real literal that can be either of floating- or fixed-point type, has value 690.0, and has enough leading zeroes to be the size of the maximum line length.	(1..120 => '0', 121..126 => "69.0E1")
$BLANKS A sequence of blanks twenty characters fewer than the size of the maximum line length.	(1..106 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
$EXTENDED_ASCII_CHARS A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.	"abcdefghijklmnopqrstuvwxyz" & "!$%?@[\]^`{}~"
$FIELD_LAST A universal integer literal whose value is TEXT_IO. FIELD'LAST.	35
$FILE_NAME_WITH_BAD_CHARS An illegal external file name that either contains invalid characters, or is too long if no invalid characters exist.	X}]!@#$^&~Y
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character, or is too long if no wild card character exists.	XYZ*
$GREATER_THAN_DURATION A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST if any, otherwise any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST The universal real value that is greater than DURATION'BASE'LAST, if such a value exists.	200_000.0

Name and Meaning	Value
$ILLEGAL_EXTERNAL_FILE_NAME1 An illegal external file name.	"bad_character*^"
$ILLEGAL_EXTERNAL_FILE_NAME2 An illegal external file name that is different from $ILLEGAL_EXTERNAL_FILE_NAME1.	"muchtoolongnameforafile" & "muchtoolongnameforafile"
$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.	32767
$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST if any, otherwise any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less than DURATION'BASE'FIRST, if such a value exists.	-200_000.0
$MAX_DIGITS The universal integer literal whose value is the maximum digits supported for floating-point types.	15
$MAX_IN_LEN The universal integer literal whose value is the maximum input line length permitted by the implementation.	126
$MAX_INT The universal integer literal whose value is SYSTEM.MAX_INT.	2147483647

Name and Meaning	Value
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER if one exists, otherwise any undefined name.	long_long_integer
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFF#
$NON_ASCII_CHAR_TYPE An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non-ASCII characters with printable graphics.	(NON_NULL)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 19 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- C32114A: An unterminated string literal occurs at line 62.

- B33203C: The reserved word "IS" is misspelled at line 45.

- C34018A: The call of function G at line 114 is ambiguous in the presence of implicit conversions.

- C35904A: The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC_ERROR instead of CONSTRAINT_ERROR as expected in the test.

- B37401A: The object declarations at lines 126 through 135 follow subprogram bodies declared in the same declarative part.

- C41404A: The values of 'LAST and 'LENGTH are incorrect in the if statements from line 74 to the end of the test.

- B45116A: ARRPRIBL1 and ARRPRIBL2 are initialized with a value of the wrong type--PRIBOOL_TYPE instead of ARRPRIBOOL_TYPE--at line 41.

- C48008A: The assumption that evaluation of default initial values occurs when an exception is raised by an allocator is incorrect according to AI-00397.

WITHDRAWN TESTS

- B49006A: Object declarations at lines 41 and 50 are terminated incorrectly with colons, and end case; is missing from line 42.

- B4A010C: The object declaration in line 18 follows a subprogram body of the same declarative part.

- B74101B: The begin at line 9 causes a declarative part to be treated as a sequence of statements.

- C87B50A: The call of "/=" at line 31 requires a use clause for package A.

- C92005A: The "/=" for type PACK.BIG_INT at line 40 is not visible without a use clause for the package PACK.

- C940ACA: The assumption that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program, is erroneous.

- CA3005A..D (4 tests): No valid elaboration order exists for these tests.

- BC3204C: The body of BC3204C0 is missing.

END

DATE

FILMED

DTIC

4/88